CUDA によるスレッドレベル並列化

Thread level parallelism in CUDA

ネットワーク情報学部　石原秀男
School of Network and Information　Hideo ISHIHARA

Keywords: CUDA, parallel computing, image processing

Abstract

NVIDIA's CUDA, Compute Unified Device Architecture, is a general purpose scalable parallel computing software platform. This platform is considered to be quite successful at programming multithreaded many core graphics processing units.This paper reviews how to build an image processing application in CUDA and investigates the performance of it.

1．　はじめに

CPU の演算性能は、動作クロックと IPC（Instructions Per Cycle）の積で決まる。クロックを上げる王道は、製造プロセスの縮小や新素材の開発だが、発熱の影響もあり近年は踊り場状態にある。マイクロアーキテクチャの面からは、パイプラインの段数を増やすという手法も採られるが、Netburst のように 30 段を超えるレベルにまで達するとストール時のペナルティが大きく実性能は上がらない。このような理由から、最近の CPU ではクロックを抑え、代わりにマルチコア化を中心とした IPC 重視のアプローチでパフォーマンスを高めようとする傾向が強くなっている。

Table.1 は、ネットブック向けからサーバ向けまで、各社の代表的な CPU ブランド[1]について内蔵するコア数と同時実行可能なスレッド数を記したものである。

CPU	Cores	Threads
Intel Atom	2	2
AMD Phenom II	4	4
Intel Core i7	4	8
Intel Xeon	4	8
AMD Opteron	6	6
Sun UltraSPARC	8	64

Table.1 Cores and Threads of various CPU

既にすべてのブランドでマルチコア化が実現されており、ハイエンドに属するものではコア当り複数のスレッドを実行できるものも多い。

一方、Table.2 に示したのは、2010 年に発売される予定の新 CPU であるが、現状と比べてコア数、スレッド数がさらに増大していることがわかるであろう。

Codename	Cores	Threads
Intel Nehalem EX*	8	16
IBM POWER7*	8	32
AMD MagnyCours	12	12
Sun Rainbow Falls	16	128

Table.2　2010 年に発売予定の CPU

表の Nehalem EX は次期 Xeon と言われる PC サーバ用 CPU だが、Intel は 2010 年春に発売するコンシューマ用ハイエンド Core i9 についてもコア数 6、スレッド数 12 とすることを発表しており、個人向け PC でも 10 を超えるスレッドが動くことになる。

また IBM の Power7 は単体では 8 コア 32 スレッドの CPU であるが、単一のパッケージに 4 つのチップを封入したマルチ・チップ・モジュールのレベルでは、SUN の Rainbow Falls と同様に 128 スレッドを処理できる。ハイエンドの世界では、100 スレッド以上を処理できる汎用 CPU、スーパマルチスレッド CPU が登場するのである。

スーパマルチスレッド CPU がターゲットとするのは、ウェブサーバやデータベースサーバなどのマーケットであるが、数年後にはコストも低下し、個人向けの PC レベルでも利用されるようになるだろう。もちろん、個人が数十ものアプリケーションを同時に実行させるという

[1] 同一ブランド内でも多様な仕様が存在するが、ここに挙げたのはそれらの上位バージョンである。

シチュエーションは考えにくいので、マルチスレッドが個別のアプリケーションの性能向上に貢献するようにならなければ意味がない。そうなると、アプリケーションを100以上のスレッドに分割するという状況が現実味を帯びてくるわけである。もちろん、マルチスレッドの効果は処理自体の並列度の高さに依存するのだが、動画処理や音声処理などを大きく変貌させる可能性がある。

ところで、あまり知られていないことだが、個人レベルでもリーズナブルな投資で100を超えるスレッドを同時実行できる環境がすでに存在している。NVIDIA社製のビデオカードで実行されるCUDAである。CUDAは、来るべきスーパマルチスレッドCPUと比べれば、用途は限定されるものの、現在のCPUレベルの上限である8とか16などという数を遥かに超えるスレッドを同時に実行することができるのである。

本稿では、CUDAについて概説するとともに、簡単な画像処理プログラムを作成し、その効果を調べてみることにする。

2. CUDAとは何か

従来、グラフィックコントローラ、グラフィックアクセラレータなどと呼ばれていたPCの描画デバイスは、座標変換や陰影処理をプログラマが自由に設定できるプログラマブルシェーダの時代になると、GPU（Graphics Processing Unit）と呼ばれるようになった。もちろんGPUは、ビデオゲームの表現力を高めることを主目的として発展してきたわけだが、Cgなどのシェーダプログラミング用言語を得ると、グラフィックに限らずゲーム内の物理演算などにも利用できるようになった。しかし残念ながら、一般的なプログラマが気楽に取り組むには無理がある。Cgの文法自体は、ほとんどC言語と同じであり、処理そのもののプログラミングはプログラマにとって難しいものではないのだが、付随するメモリ処理が独特なのである。GPUは本来3Dグラフィックスのためのデバイスであるため、Cgでは単純な数値演算であったとしても、わざわざデータを3D描画用のテクスチャデータとしてグラフィックスメモリに展開し、その参照もテクスチャとしての座標指定を通じて行わなければならない。要するにすべてのデータをわざわざ3次元の座標に当てはめる必要があるわけで、その特殊性から広く普及するには至らなかったのである。

ところで、現代のCPUは一昔前には考えられなかった素晴らしい性能を持っている。たとえばIntelのCore i7 975はクロック3.33GHzのクアッドコアプロセッサで、SIMD演算[2]を利用すればコアあたり1クロックに4個の浮動小数点演算を実行できる。つまり3.33GHz×4コア×4命令で53.28FLOPSという理論速度を持っている。1997年にチェスの世界チャンピオンを破ったことで有名なIBMのコンピュータDeep Blueは11.38GFlopsであったから、まさに10年前のスーパーコンピュータを超えるような性能なのである。

ところがデュアルポートRAMと呼ばれる特殊な高速メモリと、シンプルではあるが高度に並列化された演算器を備えるGPUは、トップクラスのCPUと比べても桁違いの演算性能を持っている。たとえばNVIDIAのGPU GeForce285GTXは、ストリーミングプロセッサと呼ばれる32ビットの演算器を240個搭載しており、GPU Review（http://www.gpureview.com）のテストで1062GFLOPSもの値を記録しているのである。

このことからもわかるように、現代のGPUは画像表示装置というだけでなく、数値演算プロセッサとしても、CPUを遥かにしのぐ第一級のポテンシャルを持っている。このGPUを汎用的な数値計算に利用しようとする技術はGPGPU（General Purpose computing on GPU）と呼ばれているのだが、CUDA（Compute Unified Device Architecture）はそのための開発環境なのである。CUDAを提供するのはGPUメーカであるNVIDIA社であり、動作するのは同社製GPU上に限られる。しかし、二大GPUメーカの一角を占めるATI（AMD）のRADEONシリーズについてもSTREAMと呼ばれる同様の環境が存在する。つまりGeForceだろうとRADEONだろうとグラフィックに限らず、数値計算にも利用できるのだ。実際、NVIDIAにはTeslaという、GPUでありながら画像出力の機能を持たない、従来の常識からすると奇妙な製品さえもある。

一般に並列コンピューティングは、異なるデータに対して同じ処理を行うデータ並列化と、同じデータに対して異なる処理を行うタスク並列化に分類される。GPUが持つ並列処理能力は、本来ポリゴンの動きを計算するために開発されたものであるから、単純な計算ルールを多くのデータに対して適用することに適している。つまりGPUは、データ並列化向きのデバイスである。その意味で用途は限定されるのだが、得意とする分野ではCPUベースのマシンとは比較にならない可能性を持っている。長崎大学の濱田助教らのグループは2009年にGeForce9800GTXなどのNVIDIA製GPUを380台接続し、158TFLOPSという日本最高速を記録したが、その製作費用はわずか3800万円だという。ピーク性能が131.07TFLOPSのSX-9（いわゆる地球シミュレータ）が180億円を超えることを考えると、驚くべきコストパフォーマンスである。また東工大のHPCであるTSUBAMEにもTeslaが使用されており、2010年に登場する次期TSUBAMEは3000TFLOPSに達する予定である。これは2009年時点の世界最速である米Cray社Jaguarの理論値2331TFLOPSを超える値であり、ハイエンドの世界でもGPGPUが注目を集めている。

[2] SIMDとはSingle Instruction Multi Data。複数のデータに対して同時に加算などの同じ種類の演算を行う。

3. CUDA のインストール

CUDA は Windows、MacOS、Linux のいずれの OS 上でも動作するが、ここでは主として Windows へのインストールについて述べよう。CUDA によるプログラム開発には、以下の 5 つが必要になる。

(1)CUDA に対応する GPU
(2)Microsoft Visual C++コンパイラ
(3)CUDA ドライバ
(4)CUDA ツールキット
(5)CUDA SDK

(1)は 256MB 以上のメモリを搭載したコンシューマ向けグラフィックスカードである GeForce8、9、200 シリーズ、もしくはプロフェッショナル向けグラフィックスカードである Quadro シリーズ、演算専用の GPU カードである Tesla シリーズのいずれかということになる。詳細な互換性リストは

http://www.nvidia.co.jp/object/cuda_learn_products_jp.html

にあるが、事実上 NVIDIA の GPU を内蔵したほとんどのグラフィックカードに対応していると考えてよい。またリストにあるビデオカードを保有していなかったとしても、ビルド時にエミュレーションモード(EmuRelease もしくは EmuDebug) を選べばプログラムのビルドと、CPU 上でのエミュレーションによる実行は可能である。つまり CUDA による開発を経験してみたいというだけなら(1)はなくても構わない。もし多少の投資を惜しまないのなら、8000 円程度で購入できる GeForce9600GT あたりのカードを入手するのが良いだろう。マザーボードオンボードの VGA などを使っているなら、普段使用しているアプリケーションのパフォーマンスも向上するはずで実用的なメリットも少なくない。なお、NVIDIA によると近い将来、CUDA が OpenCL[3]に加わり、CPU のマルチコアを利用しての並列処理なども行えるようになるらしい。

(2)については、具体的には Visual Studio .NET 2003、Visual Studio 2005、Visual Studio 2008 およびそれらに対応する Visual C++ Express Edition ということになるのだが注意が必要だ。CUDA で実際に使用されるのは VC に含まれるコンパイラ cl.exe である。後述する CUDA のバージョンが 1.0、1.1 なら VC2005 以前のコンパイラに含まれる cl.exe で問題ないが、2.0 になると VC2005 SP1 に含まれる cl.exe が必要になり、2.1 以降では VC2008 の cl.exe が必要になってしまうのだ。そういう状況なので、新たに入れるならフリーの Visual C++ 2008 Express Edition を

http://www.microsoft.com/japan/msdn/vstudio/express/

から Web インストールするのがお奨めである。Visual C++シリーズの場合には、複数のバージョンを併用しても互いに干渉することはないので、既に旧バージョンをインストール済みであっても特に気にすることはない。なお 2008 のインストール中に、Microsoft Silverlight Runtime と Microsoft SQL Server 2008 Express Edition の要否を尋ねられるが、両者とも CUDA による開発には不要である。また 30 日を超えて使用を続けたい場合には、最後に無料登録を行わなければならないが、画面に出てくる手順に従うだけなので迷うことはないだろう。なお同じページから DVD のイメージファイルをダウンロードしてオフラインインストールすることも可能で、こちらは登録の必要がないが、イメージファイルを自分で DVD に焼かなくてはならない。

(3)、(4)、(5)については

http://www.nvidia.co.jp/object/cuda_get_jp.html

からダウンロードすることができる。(3)は CUDA のためのグラフィックスドライバ、(4)はコンパイラ、ライブラリ、ヘッダファイルなどの開発ツール、(5)はサンプルプロジェクト集である。対応しているプラットホームは Windows XP 32/64 ビット、Windows Vista 32/64 ビット、Windows 7 32/64 ビット、Linux32/64 ビット、MacOS と事実上すべての OS をカバーしているが、前述したように 1.0、1.1、2.0、2.1、2.2、2.3 の各バージョンが存在するので、インストール済みのコンパイラのバージョンに合わせて、自分の環境にあったものを選ばなくてならない。今回は、Visual Studio 2005 がインストール済みのマシンを利用したので、CUDA1.1 をインストールすることとした。インストールしたマシンは以下のスペックを持つ自作機である。

MB : DG31PR
CPU : Intel Core2 Quad Q9550(2.83GHz)
GPU : GeForce8600GT
Memory : 1MB×2 (DDR2 667MHz)
HDD : Seagate Barracuda ST3250310AS (250GB)
OS : Windows XP 32bit SP3
Compiler : Visual Studio 2005

[3] アップルが提唱した、マルチコア CPU、GPU などを異種させた環境での並列処理のための開発環境を作成しようというプロジェクト。

手順としてはまず、CUDAドライバとして、Windows XP用CUDAサポートNVIDIAドライバ169.21（169.21_forceware_winxp_32bit_international_whql.exe）をインストールする。なお、以前は専用のCUDAドライバが必須であったが、現在ではNVIDIAのグラフィックスドライバであるForceWareに機能が統合されており、185.xx以降のバージョンのドライバを使用していれば新たにドライバをインストールする必要はないとのことである。

次にCUDAツールキットとしてWindows XP用CUDAツールキットバージョン1.1をインストールする。デフォルトではC:¥CUDAにインストールされるが、特に理由がない限りそのままにしておくのがよいだろう。なお、64ビット版のOSを使用している場合には注意が必要だ。実はVisual C++ 2008 Express Editionには、(抜け道はある)64ビットアプリケーションを作成できないという制限[4]がある。そのためか、別のVista64ビットマシンに64ビット用ツールキットをテストしたところ、全く動作しなかった。そこで、Vista64ビットに32ビット用ツールキットをインストールしてみたところ、特に問題も発生せず、簡単なアプリケーションの作成程度では不具合は出なかった。64ビットOSにインストールするときには、この方法を試してみる価値はあるだろう。

最後に、CUDA SDKとしてWindows XP用CUDA SDKバージョン1.1をインストールする。途中で1.1以上のツールキットが必要だというメッセージが出るが、すでにインストールしてあれば気にすることはない。

以上が終了したら、CUDA SDKのインストール先[5]にあるサンプルプロジェクトをビルドして実行し、インストールの可否を確認する。

たとえば、deviceQueryというサンプルプロジェクトなら、同名のフォルダ内のプロジェクトファイルdeviceQueryをダブルクリックするとVisual Studioが起動するので、メニュー直下にあるドロップダウンボックスから構成マネージャーでDebugかReleaseを選び、その左横にある矢印▸をクリックすればよい。もしここでEmuDebugかEmuReleaseを選べば、エミュレーションモードとなりNVIDIAのGPUが存在しない環境でもビルドや実行を行うことが可能である。

インストールが成功していれば、プロジェクトがビルド、実行されdeviceQueryの結果としてFig.1のようなものが表示されるはずである。

```
There is 1 device supporting CUDA

Device 0: "GeForce 8600 GT"
  Major revision number:                         1
  Minor revision number:                         1
  Total amount of global memory:                 268107776 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1188000 kilohertz

Test PASSED

Press ENTER to exit...
```

Fig.1 Result of "deviceQuery"

deviceQueryはその名が示すとおり、搭載されているGPU（このケースではGeForce8600GT）の情報を表示するプログラムである。Major version number 1、Minor version number 1は、インストールされているCUDAのバージョンが1.1であることを意味している。

CUDAでは従来のアプリケーションと比較すると非常に多くのスレッドを扱うため、スレッドをまとめたものをblock、blockをまとめたものをgridと呼びスレッドの管理を階層的に行っている。Maximum number of threads per blockの512は各blockあたりの最大スレッド数を、Maximum sizes of each dimension of a blockの512 × 512× 64は、3次元的に番号付けされるblock内のthreadsサイズの最大値が、x成分およびy成分について512、z成分について64である[6]ことを、Maximum sizes of each dimension of a gridの65535 × 65535× 1は、3次元的（実際には2次元）に番号付けされるgrid内のblockサイズの最大値がx成分およびy成分について65535、z成分について1であることを表している。つまりこの場合は、blockあたり最大512スレッドで、

[4] 64ビットOS上で動作しないという意味ではない。64ビットOS上で32ビットのアプリケーションを作成することは問題なく可能である。

[5] デフォルトではC:¥Program Files¥NVIDIA Corporation¥NVIDIA CUDA SDK¥projects

[6] 同時にこの値を取れるわけではない。あくまでもblock内のスレッドの最大数はMaximum number of threads per block で定められた512である。

その block を最大 65535 × 65535 個まで管理できることになる。もちろん、実際に同時実行されるスレッドの数は GPU 内の演算器の数に依存する。8600GT に搭載されているストリーミングプロセッサは 32 個[7]であるから、32 スレッドが同時実行できる。しかしながら、CPU 側から見た場合、一つのストリーミングプロセッサで 1 クロックあたり同一命令が 4 回実行されるとのことなので、128 命令が実行されることになる。

4. CUDA アプリケーションの実際

CUDA によるアプリケーションの作成は、SDK にサンプルプロジェクト template があるので、それをベースに行えばよい。

template をそのまま書き換えてしまうこともできるのだが、ここでは prog1 という名称のプロジェクトを作成することを例として手順を説明しよう。

(1)SDK のインストールフォルダに prog1 という名称のフォルダを作成

(2)template フォルダにある

template.sln	ソリューションファイル
template.vcproj	プロジェクトファイル
template.cu	メインプログラムのソース
template_gold.cpp	CPU 側関数のソース
template_kernel.cu	GPU 側関数のソース

をすべて pro1 フォルダにコピー。

(3)コピーしたファイルを
prog1.sln
prog1.vcproj
prog1.cu
prog1_gold.cpp
prog1_kernel.cu
とリネーム。

(4)prog1.sln と prog1.vcproj 中の文字列"template"をすべて"prog1"と変更。(テキストファイルなのでメモ帳などで開いて置換すればよい。)

(5)prog1.vcproj をダブルクリックし VC2005 を起動。

(6)メニューから
ビルド(B)...ソリューションのビルド(B)
を実行。

(7)メニューから
デバック (D) ...デバックなしで開始 (H)
を実行。

```
Processing time;0.0091662(ms)
Test PASSED
```

などと表示されれば成功。

(4)が面倒な場合には、(3)、(4)を省略して替わりに(5)で template.vcproj を選んでしまっても特に支障はないが、その場合には以下の"prog1"を"template"と読み替えてもらいたい。

さてプロジェクト prog1 で、アプリケーションのソースに該当するのは、分割コンパイルと保守性のために分けられた prog1.cu, prog1_gold.cpp, prog1_kernel.cu の 3 ファイルであり、それぞれの役割は以下のようになっている。

prog1.cu はプログラムのメイン部分のソースで、pro1_gold.cpp と、prog1_kernel.cu はここから呼び出される関数のソースである。prog1.cu 自体は基本的に C/C++言語にしたがっているが、CUDA 特有の文法を含むため拡張子は cu に変更されている。

prog1_gold.cpp は prog1.cu から呼び出され CPU 上で実行される関数のソースで、純粋な C/C++言語で書かれているので拡張子も cpp である。本稿で扱っているような数十から数百行程度のプログラムでは、わざわざ通常の関数を別ファイルに分ける必然性はなく prog1.cu 内に書いても構わない。

prog1_kernel.cu は GPU で実行される関数のソースであり、CUDA ではそのような関数をカーネル関数と呼ぶことからこの名が付けられている。拡張子は prog1.cu と同様な理由から cu である。この prog1_kernel についても本稿で扱う程度のプログラムでは別ファイルにするメリットはあまりない。結局のところ、すべての関数を prog1.cu に含めることとし、prog1.cu だけを必要に応じて書き換えればよいのである。

以下では、1 から 100 までの自然数の和を求めるという簡単な計算を例として、CUDA プログラミングの書き方を解説しよう。すでに述べたように CUDA が向いているのはデータ並列化である。行おうとしているのは 100 個の自然数を足し合わせることであるから、10 個の自然数の和を求めるという処理を 10 回行い、最後にそれらの結果を足し合わせればよい。分割の方法はいろいろ考えられるが、1 から 10 までの和、11 から 20 までの和、・・・、91 から 100 までの和というように分けるのが自然だろう。もちろん、最後にこれらを足し合わせる作業も CUDA で行えるが、この例では CPU 上で行う方が簡単である。以上の考え方に基づき作成したのが List.1 である。

[7] ハイエンドカードではもっと多い。たとえば GTX295 は 480 個、GTX280 は 240 個である。

```
#include <stdlib.h>
#include <stdio.h>
#include <cutil_inline.h>

__global__ void addKernel(int *d_idata, int *d_odata){
    int min = d_idata[threadIdx.x] ;
    int sum = 0 , i ;

    for( i = min ; i < min + 10 ; i++)
        sum += i;

    d_odata[threadIdx.x] = sum;
}

int main( int argc, char** argv  )
{
    unsigned int mem_size = sizeof(int)*10 , i ;
    int h_idata[10] , int *d_idata , *d_odata ;

    for(i=0;i<10;i++)
        h_idata[i]=i*10+1;

    cutilSafeCall( cudaMalloc( (void**) &d_idata, mem_size) );
    cutilSafeCall( cudaMalloc( (void**) &d_odata, mem_size) );
    cutilSafeCall( cudaMemcpy( d_idata, h_idata, mem_size, cudaMemcpyHostToDevice) );

    dim3 grid(1,1,1);
    dim3 threads(10,1,1);

    addKernel<<< grid,threads>>>(d_idata,d_odata);

    cutilSafeCall( cudaMemcpy( h_idata,d_odata,mem_size,cudaMemcpyDeviceToHost) );

    int sum = 0;

    for(i=0;i<10;i++)
        sum += h_idata[i];
    printf("¥n%d",sum);

    cutilSafeCall(cudaFree(d_odata));
    cutilSafeCall(cudaFree(d_idata));

    cutilExit(argc, argv);
    return 0;
}
```

List.1 Sum of the Natural Numbers from 1 to 100

List.1 には二つの関数が含まれているが、main は CPU 上で実行される通常の main 関数、addKernel は GPU 上で実行されるカーネル関数である。

addKernel の修飾子__global__は CPU から呼び出され GPU 上で実行されるカーネル関数であることの宣言であり、GPU から呼び出され GPU 上で実行される関数については__device__という修飾子を用いることになっている。カーネル関数については、void 型しか許されない、再帰が使えない、静的変数が使えない、CPU 側の変数には（たとえグローバル変数であっても）アクセスできない、などの制限はあるがそれらの点を除けば通常の関数とあまり違いはない。

引数 d_idata は引数を渡すための配列である。このケースでは10個の addKernel が同時に動作するわけだが、それぞれのスレッドには 0 から 9 までの番号が付けられており、n 番目のスレッドには n*10+1 から n*10+10 までの和を計算させることになる。そこで d_idata[n]に n*10+1 を代入して引数として渡している。各スレッドは自分のスレッド番号を threadIdx.x で参照できる[8]ので、min = d_idata[threadIdx.x]とすれば目的の部分を取り出せる。もう一つの引数 d_odata は各スレッドの計算結果を返すために用意した配列であり、n 番目のスレッドの結果は d_odata[n]に代入して返すことになる。

カーネル関数のメモリモデルには、各スレッド固有のレジスタとローカルメモリ、ブロック内のスレッドで共有されるシェアード（shared）メモリ、全スレッドで共有されるグローバルメモリ、コンスタントメモリ、テクスチャメモリがある。Fig.1 で表示されていた

global memory:	268107776 bytes
constant memory:	65536 bytes
shared memory per block	16384 bytes
registers available per block:	8192 bytes

は、これらの容量である。

addKernel では、二つの引数 d_idata、d_odata としてグローバルメモリを使用し、カーネル内部の min や sum などの変数にはローカルメモリを使用している。プログラミングガイドによるとこれらのメモリへのアクセスは非常に遅く、パフォーマンスを求める場合にはシェアードメモリを使用することを推奨している。つまり、addKernel の場合には、d_idata をシェアードメモリにコピーし、それを用いて計算を行った上で、d_odata へコピーすべきということである。また、スレッド内で宣言したメモリはシェアードメモリと同等以上に高速なレジスタへと優先的に配置されることようなので、その点にも留意すべきであろう。もし、カーネル関数内でシェアードメモリを宣言する場合には変数宣言時に__shared__と修飾子を付けるだけでよい。

ところで並列プログラミングには必須の同期であるが、cudaMemcpy の直前で同期が取れることが保証されているのでこのケースでは特に気にする必要はない。任意の場所で同期を取りたいときは、そこに

```
_syncthreads();
```

の一文を記述すればよい。

main 側では、まず i 番のスレッドに渡すためのデータ i*10+1 を計算し、h_idata に格納している。次の cutilSafeCall[9]は CUDA 関数を呼び出す関数である。_DEBUG が define してあれば、関数のエラーメッセージを表示するという機能を持っているが、本質的に必要なわけではなく関数を直接呼び出しても構わない。呼ばれている cudaMalloc は GPU 上での malloc でこの例では、d_idata、d_odata それぞれについて mem_size バイトのメモリを確保している。cudaMalloc の書式は

```
cudaMalloc(void **ptr , size_t size)
  ptr  GPU メモリアドレスへのポインタ
  size 確保するメモリのサイズ
```

である。

CPU から GPU へのデータ渡しは形式的には引数で行われているのだが、それだけでは GPU から CPU 側の変数にアクセスすることはできず、明示的に値を渡さなければならない。次の cudaMemcpy では、Host(CPU)側の変数 h_idata を Device（GPU）側の変数 d_idata にコピーしている。cudaMemcpy の書式は

```
cudaMemcpy(void *ptr1 , void *ptr2 ,
                        size_t size, int mode)
  ptr1    転送先アドレス
  ptr2    転送元アドレス
  size    転送サイズ
  mode    転送モード
    cudaMemcpyHostToDevice なら CPU から GPU
    cudaMemcpyDeviceToHost なら GPU から CPU
```

であり、転送モードで CPU−GPU 間のコピー方向を制御する。

次の dim3 は 3 次元のデータ型として grid と threads の二つの変数を宣言している。CUDA ではスレッドの集まりをブロック、ブロックの集まりをグリッドと呼ぶが、変数 grid はグリッドに、変数 threads はブロックに対応

[8] スレッドを 1 次元にしている場合。2 次元の場合には threadIdx.x+blockDim.x*thread.Idx.y。

[9] バージョンによっては CUDA_SAFE_CALL（）が使われる。

している。grid を(1,1,1)と宣言したことは、グリッドがただ一つのブロックからなっていることを、threads を(10,1,1)としたことは、ブロックがx方向には10個、y、z方向には1個のスレッドからなっていることに相当する。つまりこのアプリケーションは全体で10個のスレッドからなり、各スレッドはx成分として0から9までの番号を持つことになるわけで、それゆえ addKernel 内では threadIdx.x として0から9までの値が存在するのである。

この例では全体を一つのブロックとしたが、プログラミングガイドでは、搭載されているストリーミングプロセッサ数の2倍以上のブロックに分割することがパフォーマンスの面から望ましいとしている。またブロックあたりのスレッド数についても64の倍数とすることが推奨されている。このように、CUDA は本来非常に多くのスレッドを前提に作られているのである。

実際にカーネル関数を起動するのは、次の addKernel である。<<<grid,threads>>>で定められた10個のカーネル関数が同時に起動され、引数として()内の d_idata、d_odata が与えられる。一般にカーネル関数の呼び出しの書式は

func<<<dim1,dim2,Ns,S>>>(param1,param2,・・・)
- func　呼び出されるカーネル関数名
- dim1　グリッドのサイズ
- dim2　ブロックのサイズ
- Ns　各ブロックに割り当てるシェアードメモリのバイト数（省略可）
- S　ストリーム（省略可）
- param1,param2・・・　引数の並び

である。ここで Ns はシェアードメモリのサイズであるが、カーネル関数内で__shared__ int a[Ns]などと割り当てる替わりに Ns を設定し、__shared__ int a[]とすることができる。また S はストリームと呼び、カーネルに関して複数のフローを存在させるときに、その属する流れを指定するためのものである。Ns、S については使用しないならば省略してもよい。

次の cudaMemcpy は cudaMemcpyDeviceToHost を指定することによって、前とは逆に各スレッドの計算結果を d_odata から h_idata へと戻している。

その後は、CPU 上ですべてのスレッドについて h_idata を足し合わせて総和を求め、cudaFree で malloc したメモリを解放し cutilExit で終了している。

以上のことからわかるように CUDA アプリケーションは

(1)GPU 上にメモリ確保
　　cudaMalloc を使用
(2)CPU から GPU へ引数データを転送
　　cudaMemcpy を使用
(3)カーネル関数を呼び出し GPU 上で演算
(4)GPU から CPU へ演算結果を転送
　　cudaMemcpy を使用

の順に構成すればよい。これを理解していれば独自のアプリケーションを作成するのも難しくないだろう。

5. アプリケーションの性能

ここでは、前章で扱ったものよりも現実的なプログラムを作成し、実際のアプリケーションにおける CUDA の効果を調べてみることにする。使用している GPU がローエンドの 8600GT であるため、絶対的な処理時間には大きな意味はない。しかし、処理の分割数（＝スレッド数）と実行時間の関係を調べれば、CUDA の可能性を検証することができるはずである。

具体的なプログラムの内容は、640×480 ドットの 24 ビットカラー画像を 4×4 ドットのブロックサイズで平均しモザイク化するものである。この処理はブロックごとに独立して並列に実行できるため、CUDA には最適なものと言えるだろう。並列化の手法としては、画像全体を同サイズの領域に分割し、各分割に対して一つのスレッドを割り当てる。分割数としては、1、2、3、4、5、6、8、10、12、15、20、24、30、40、60、120、240、480 を選び、それぞれに対する実行時間を測定する。たとえば 480 個の領域に分割する場合には、一つの領域は 160×4 ドットの大きさとなり 40 個のモザイク化ブロックを含むことになる。

分割数と実行時間の関係は、十分な数の演算装置があり理想的な並列処理が行われれば、反比例になるはずである。8600GT の場合にはクロック当り 128 命令が同時実行可能であるから、128 分割までは実行時間がスレッド数に反比例して減少することが期待される。

ところが一般的な CPU では、その予想が成り立たないこともある。

CPU	Xeon5345×2	Core2 E6700
2threads	50.0%	73.5%
4threads	25.0%	64.3%
6threads	16.9%	58.3%
8threads	17.0%	56.9%
10threads	16.9%	56.8%

Table.3 number of threads and execution time

Table.3 は、筆者が二つの CPU について行った[10]スレ

[10] 文献 5 では、与えられた区間に存在する素数の数を求めるという処理について調べた。

ッドの分割数と実行時間の測定結果である。表で Xeon5345×2 の 2 スレッド時の 50.0%という値は、2 スレッドに分割し並列実行した場合の処理時間が 50%になったということを表している。この Xeon の場合には実質 8 コアなので、理論的な最高値は 8 スレッドのときの 12.5%ということになるが、そこに至る経過も含めてかなり予想に近い結果と言えるだろう。

一方、デュアルコアの Core2 Duo では 2 スレッドに分割しても実行時間は 70%超にしか短縮できず、50%台の性能を得るためには 6 スレッド以上への分割を行わなければならない。アプリケーションと同時に OS の常駐プログラムなどもスケジューリングされるため、スレッド数が少ない状況ではアプリケーション側の CPU 時間が相対的に低下してしまうことがその理由かも知れない。もちろん、望ましいのは Xeon と同様の結果が得られることで、そうなればより多くのストリーミングプロセッサを持つ最新の GPU を用いればさらに高い性能が期待できることになる。

作成したプログラムはかなりのボリュームになるため、基本的な考え方だけを述べ、120 スレッドに分割する場合のカーネル関数のみを巻末の Appendix に記載した。

プログラム全体の流れとしては以下のようになる。

(1) 画像ファイルを CPU のメモリに読み込む。
(2) cutCreateTimer(&timer)で時間測定のためのタイマを作成
(3) cutStartTimer(timer)でタイマをスタート
(4) cudaMalloc で画像データを渡すための 921600[11] バイト分のメモリを確保
(5) cudaMemcpy で画像データを GPU へ転送
(6) grid(1,1,1)、threads(スレッド数,1,1)、としてカーネル関数を呼び出す
(7) cudaMemcpy でモザイク処理されたデータを CPU へ転送
(8) cudaFree でメモリを解放
(9) cutStopTimer(timer)でタイマをストップ

処理時間が計測されるのは、(4)から(8)の間で、fscanf で行っている画像ファイルの読み込みなどは含まれていない。実際のプログラムでは、この後ウィンドウを開き、確認のためモザイク化された画像を表示しているのだが CUDA とは無関係なのでここでは省略する。

モザイク化の計算は、(6)で呼び出されるカーネル関数の内部で行っている。具体的には、画面左下を起点として各画素について 1 次元配列に青、緑、赤の順に格納されている画像データから、画面上で隣接する 4×4 ドットの領域（16 ドット）分を取り出し、R、G、B 各成分についての平均値を求め、元の配列の相当する 16 ドットの位置に書き戻す。Appendix に掲載した 120 スレッドの場合の分割は、画面下から 4 ライン単位ごとに 640×4 ドットの領域を一つのスレッドとしている。

以上のようにして、スレッド分割ごとの処理時間を測定した結果が Table.4 である。

threads	time(ms)	rate(%)
1	370.06	100.0
2	181.96	49.2
3	125.51	33.9
4	97.27	26.3
5	80.03	21.6
6	69.61	18.8
8	58.03	15.7
10	49.25	13.3
12	43.64	11.8
15	38.48	10.4
20	34.22	9.2
24	32.36	8.7
30	32.13	8.7
40	26.65	7.2
60	20.80	5.6
120	15.70	4.2
240	12.51	3.4
480	11.74	3.2

Table.4　execution time in CUDA

また、Table.4 のスレッド数と実行時間をグラフにしたものが Fig.2 である。

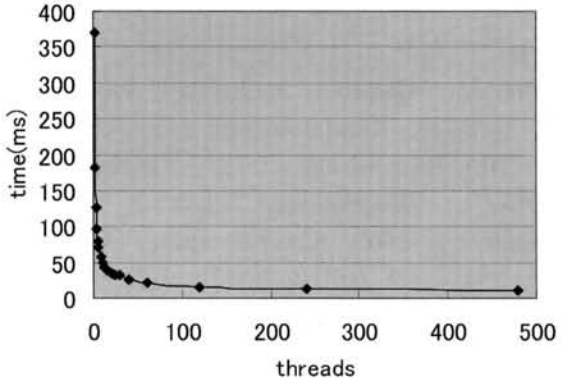

Fig.2 threads and execution time in CUDA

Table.4 を見れば 480 スレッドに分割した場合に最高性能が得られ、処理時間は 1 スレッドの場合の 3.2%程度、つまり 31.5 倍の性能が得られていることがわかる。

[11] 640×480×3

絶対的な時間としても 370ms も要していたものが、10ms 少々まで短縮されるわけで、リアルタイム処理が可能なレベルまでスピードアップしたと言えるだろう。

また、Fig.2 からも明らかなように 8600GT の同時実行可能数を超える 480 スレッドに至るまで、スレッド数と実行時間の間には逆比例的な関係がある。しかしながら、ほぼ反比例と言えるような関係が成立するのは 10 スレッド程度以下の範囲に限られ、それ以上のスレッド数になると時間短縮の割合は小さくなる。この例では CPU−GPU 間のデータ転送に 10ms 程度を要しているものと考えられるので、それを除けばスレッド数と実質的な計算時間についてはかなり理想的な関係が成立していると言えるだろう。

6. おわりに

本稿では CUDA による並列処理の実際について述べた。アプリケーションを素直に並列化しただけで、数 10 倍ものパフォーマンスが得られるのは、従来の CPU の世界からは考えられないことである。今回取り上げたモザイク化処理では、GPU へのメモリ転送の負荷が重いため、さらなるスレッド分割を行ってもこれ以上の結果は望めない。しかし複雑な計算を伴う処理については数 100 倍もの性能が得られるケースもあるだろう。

誰かの言葉に「計算機が数倍速くなっても快適になるだけだが、数 100 倍速くなれば世界が変わる。」というのがあった。CUDA に代表される超並列化は、まさにそれを予感させるパラダイムなのである。

謝辞

本研究は、著者が平成 20 年度中期研究中に行われたものである。この場を借りて研究の機会を与えてくれた大学に感謝の意を表したい。

参考文献

[1] NVIDIA Corporation, 2008, GPU Programming Guide GeForce 8 and 9 series
[2] NVIDIA Corporation, 2009, Getting Started NVIDIA CUDA Development Tolls 2.2
[3] Top500 Org, 2009, Super Computer List
[4] 長崎大学, 2009, GPU クラスタによる計算がゴードンベル賞を受賞
[5] 石原, 2006, スレッドレベル並列性とプロセッサ性能, 専修大学ネットワーク&インフォメーション No.9
[6] 石原, 2008, クアッドコアプロセッサの性能,専修大学情報科学研究所所報 No.69

Appendix

```
__global__ void ON_GPU(unsigned char *data){
    int sum, lp;
    BYTE *org,*bits;

    for(lp=threadIdx.x;lp<threadIdx.x+120;lp++){
        org=data+640*3*4*lp;
        for(bits=org;bits<org+640*3;bits+=4*3){
            // BULE
            sum = 0;
            for(int i=0; i<4 ;i++)
                for(int j=0; j<4 ;j++)
                    sum += *(bits+i*3+j*640*3);
            sum /= 16;
            for(int i=0; i<4 ;i++)
                for(int j=0; j<4 ;j++)
                    *(bits+i*3+j*640*3)
                        =(unsigned char)sum;
            // GREEN
            sum = 0;
            for(int i=0; i<4 ;i++)
                for(int j=0; j<4 ;j++)
                    sum += *(bits+i*3+j*640*3+1);
            sum/= 16;
            for(int i=0; i<4 ;i++)
                for(int j=0; j<4 ;j++)
                    *(bits+i*3+j*640*3+1)
                        =(unsigned char)sum;
            // RED
            sum = 0;
            for(int i=0; i<4 ;i++)
                for(int j=0; j<4 ;j++)
                    sum += *(bits+i*3+j*640*3+2);
            sum/= 16;
            for(int i=0; i<4 ;i++)
                for(int j=0; j<4 ;j++)
                    *(bits+i*3+j*640*3+2)
                        =(unsigned char)sum;
        }
    }
}
```

List.2 Kernel function to tessellate